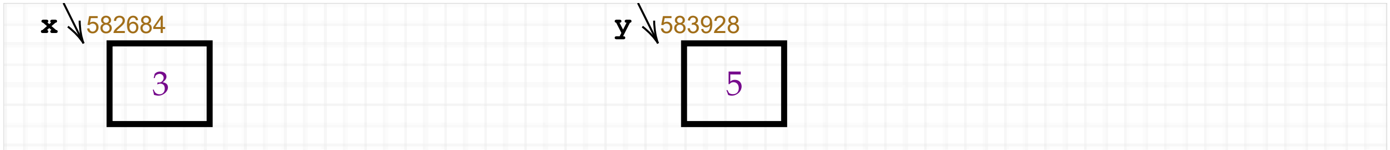
CSE305 Week 6 on Variables: Their Names, Bindings, Values, and Scopes

Picking up in Sebesta, chapter 5:

Assignments and Bindings

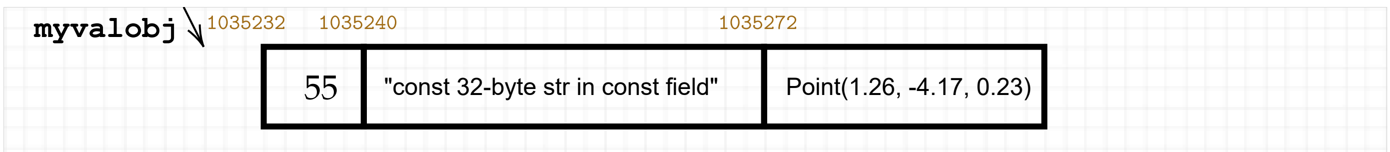
The most important fact to realize about bindings---both regular and static---is this:

- A value assignment **x = y** does not change the storage-object binding of **x** (nor of **y**).

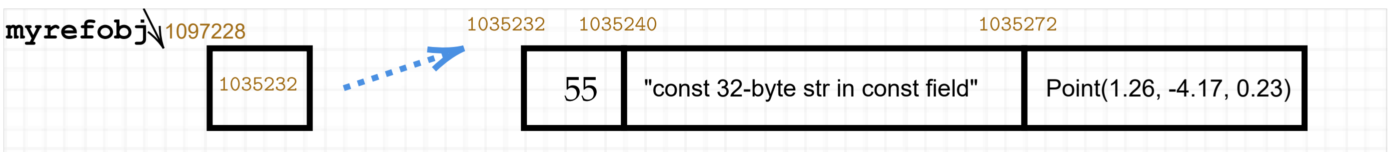


What happens is that the variable y is **automatically dereferenced** to get the current value of its storage object (here shown as `5`). The value is copied over the former value of the storage object bound to **x**. Neither of the bindings---the short arrows---changes.

The second most important fact is that a **reference**---in Java or Python or Scala especially---is almost universally implemented via a pointer object. That is, a single reference variable is bound to a tandem. The contrast with a *pure value object* (available in C++ and Scala) is:

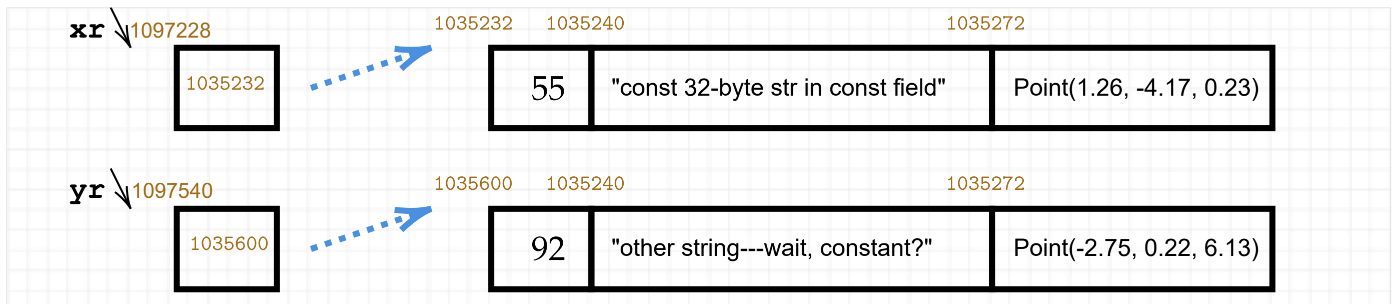


versus



OCaml makes this implementation explicit by having an `'a ref` object be separate from whatever type `'a` object it references. Syntactically, `ref` is like `list`, just another (lowercase!) type constructor. So you can have `int ref`, `float ref`, `string ref`, `int list ref`, etc. Regardless of implementation details, references are always implemented to bring about this key fact:

- A reference assignment xr = yr behaves like a value assignment of the references. As such, it does not change any actual bindings, but has the indirect effect of doing so.



The actual data does not get copied---which is a big point of "reference semantics". (So this does not violate the assertion about the string field being constant.)

- The binding of the variable **`xr`** to its reference storage object did not change.
- But the change in the reference value (which the address of the object referenced by **`yr`**) gives the indirect effect of **`xr`** being bound to the new object.

The third important point and question is:

- If you want to give the semantics that a name is bound to a *value*, as in a mathematical formula, then you cannot freely allow assignment statements. (Other than initializations, that is; I prefer calling the others ***re-***assignment statements.)

When reading OCaml in particular, we want to see whether implementations can afford to bind directly to values.

Changes in Bindings

The same name may possibly reference different storage objects as the program is run---but not simply by assignment statements. Changes in actual bindings occur when code is "reactivated". The text (later) uses the term **subprogram** as a catch-all for: function, method, procedure, subroutine...whatever programming languages call code that can be repeatedly called.

1. Any creation of a new **activation frame** for the same **subprogram** re-binds all the parameters and variables mentioned in its body to new storage objects created in the frame. (Older editions of Sebesta's text called this **activation binding**.)
2. Virtual methods (via **dynamic dispatch**) in OOP languages.
3. Dynamic creations of new functions---not just in functional languages like OCaml but when creating or modifying "function pointers" in C++ (or "delegates" in C#).
4. Passing functions as parameters---things get complicated enough that we'll deal with this fully later in section 9.6 of the text.

Here's a question: is match-with in OCaml, or match-case in Scala, an example of dynamic binding?

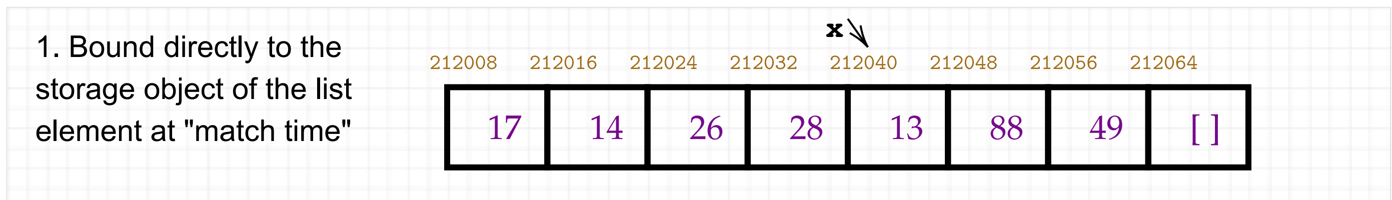
This matters when the body of the match is executed more than once, which typically involves recursion. Let's re-visit list and tree examples of a kind we've seen:

```
let rec sumList2 ell = match ell with
      [] -> 0
   | x::rest -> x + sumList2(rest) + x;;
```

This computes double the sum; we've put **x** "before" and "after" the recursive call to help visualize what happens to the storage object bound to the name **x** at both times. The issue of evaluation order (time-wise) will reappear in Chapter 7, but we can take it as read from Chapter 3 that the first `+` involved in "`x + sumList2(rest)`" is evaluated first. Some observations:

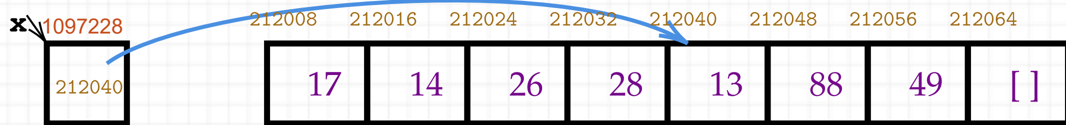
- A new storage object for "**x**" is being created at each level of recursion. If we had a single storage object whose values were overwritten in each call, then the previous value would not survive to be added "afterward" in the "`+ x`" part.
- In the activation frame for each level, **x** refers to an element of the list.
- One and only one object **x** is created in each frame. That is, in each recursive call, the three occurrences of **x** refer to the same element which is already in the list `ell`.

The main question is: Is **x** bound directly to the list element at the time it is matched up along with the rest of the list? Or does it get a copy of the list element? There are actually three possibilities:



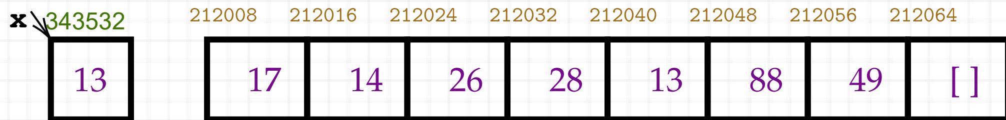
This means that **x** is bound the the same place as **`ell`** in this recursive call. If **`ell`** is the empty list, it doesn't hurt to alias **x** to **`ell`** then too. So **x** can be bound at activation time in this case too.

2. Bound to an explicit list pointer.



(Again there's a 2'. in which only the values from 13 on are present.)

3. Bound to an ordinary storage object that gets a copy of the element.



The answer in all three cases tends to say: *x is bound at activation time not "match time".* [Voiceover: It used to be axiomatic that Lisp was implemented in style 2, but with sophisticated compilers now, I have no idea---it can be an implementation choice not a language design matter.]

Now how about our tree code. Let's use the version from the recitation slides:

```
type 'a btree = Empty
  | Node of 'a btree * 'a * 'a btree
exception Duplicate;;
let initTree() = Empty;;              (* A fn with "unit" argument.*)

let rec insert (item, t, lessThan) = match t with
     Empty -> Node(Empty,item,Empty)  (*one-node tree holding item.*)
   | Node(ell,x,r) ->
       if lessThan(item,x) then       (*tree with item inserted into ell.*)
          Node(insert(item,ell,lessThan),x,r)
       else if lessThan(x,item) then  (*tree with item inserted into r.*)
          Node(ell,x,insert(item,r,lessThan))
       else raise Duplicate;;         (* Per trichotomy, item = x. *)
```

Here, "**x**" refers to the elements stored in the nodes that are traced until reaching a place where a new `Node` with the presented `item` can replace an `Empty` subtree.

I argue that as with the list recursion, this is not anything more "magical" than *activation binding*. Even if we want to say that **x** is bound directly to the value in the node when it is matched, it is probably most readily implemented using a pointer under-the-hood.

Interlude: About the lessThan pass-in and storage for functions.

For a `string btree`, here are some `lessThan` functions we could pass in:

```
let ltlen (s1,s2) = (String.length s1) < (String.length s2);;
let ltascii(s1,s2) = s1 < s2;;
let ltalpha(s1,s2) = (String.lowercase s1) < (String.lowercase s2);;
```

To create the tree from a more-conveniently inputted list:

```
let rec list2tree(ell,t,lessThan) = match ell with
      | [] -> t
      | x::rest -> list2tree(rest, insert(x,t,lessThan), lessThan);;
```

Some lists to try:

```
let ell1 = ["I";"shoot";"the";"Hippopotamus";
"with";"bullets";"made";"of";"platinum"];;
let ell2 = ["Because";"if";"we";"used";"leaden";"ones";
"his";"hide";"would";"surely";"flatten";"'em"];;

let ell3 = ["Because";"because";"because";"because";"because";
"Because";"of";"the";"wonderful";"things";"he";"does!"];;

let t1 = list2tree(ell1, initTree(), ltalpha);;
```

We can make a way to "pretty-print" the tree, but rotated sideways. This can work for a tree of any item type 'a, but needs passing in another function: one to print the item as a string.

```
let rec spaces(n) = if n > 0 then " "^spaces(n-1) else "";;  (*used by printTree.*)

(** Pretty-prints tree t "sideways", with right subtrees topmost.
 *)
let rec printTree(t, indent, offset, item2string) = match t with
    Empty -> print_string("")
  | Node(ell,x,r) ->                             (* Print right subtree topmost. *)
        printTree(r,indent+offset,offset,item2string);
        print_string(spaces(indent)^"()"^(item2string x)^"\n");
        printTree(ell,indent+offset,offset,item2string);;

printTree(t1, 4, 4, fun x->x);;
```

These are both examples of OCaml having "First Class Functions". They would work similarly in

Python and Scala too. You don't even have to say "function pointer" (C/C++) or wrap it in a "function object" (Java) or "delegate" (C#)---just pass the name of the function and it's good to go.

- To what kind of storage object is the name of the function bound?

When subprograms are not first-class, the answer is easy: there is an address where the code for the subprogram is stored and all invocations of the subprogram are bound to that address for the lifetime of the program.

The names `lessThan` and `item2string` in these functions, however, get code that is passed in from multiple places. When are they bound?

- The simple boring answer is: "at activation time."

The less-boring and more complete answer will come in chapters 9--10.

Besides passing functions as parameters, however, we can also create functions on-the-fly. The simplest way to do this is as a special case of another function with a value filled in.

```
let plus (x,y) = x + y;;            (** type int * int -> int **)
let add3 y = plus(3,y);;            (** type int -> int **)
```

This is neater to do if we write the arguments in **curried** form:

```
let plusc x y = x + y;;              (** type int -> int -> int **)
let add3 = plusc 3;;                 (** type int -> int        **)
```

Becuase the function arrow is right-associative in a *type expression* (!), the OCaml-reported type `int -> int -> int` groups as `int -> (int -> int)`. That is, `plusc` counts as a function that maps an integer to a function of type `int -> int`. The next line then initializes the name `add3` to the body of that function. How and where in the implementation is it bound?

Same question if we "curry" the `insert` function (or had written it that way to begin with):

```
let insertCurry lessThan (item,t) = insert(item, t, lessThan);;
let insa = insertCurry ltalpha;;
```

Wow, we've actually created code by passing in code. We can then use it to write code that is both cleaner and safer:

```
let rec list2treeAlpha (ell,t) = match ell with
   | [] -> t
   | x::rest -> list2treeAlpha (rest, insa(x,t));;
```

```
let t1 = list2treeAlpha (ell1, initTree());;
let t2 = list2treeAlpha (ell2, t1);;
printTree(t2,4,4,fun x->x);;   (* could curry/closure this too. *)
```

Functions like `add3` and `insa` are commonly called **taking a (simple) closure of** the more-general functions they specialized. Formally, the term **closure** includes also the need to determine the entire **referencing environment** of the passed-in code, when the passed-in code might involve using variables that it does not declare. (Peek ahead to section 9.6, pages 393-394 in the online 12th edition, if you wish to see why this is not simple.) Well, it's high time we start talking about referencing environments, beginning with the notion of lexical scope.

Type Bindings (from Sebesta 1998)

1. How is a type specified?
2. When does the binding take place?

Def: An **explicit declaration** is a program statement used for declaring the types of variables.
Def: An **implicit declaration** is a default mechanism for specifying types of variables (usually at the first appearance of the variable in the program).

FORTRAN, PL/I, BASIC, and Perl provide implicit declarations.
Advantage: writability
Disadvantage: reliability (less trouble with Perl since Perl programs are supposed to be short)

Dynamic Type Binding (APL example):
LIST <- 2 4 6 8 (an integer array)
LIST <- 17.3 (a single float)
Main disadvantage: Type error detection by the compiler is difficult.

Type Inferencing (ML, Miranda, Haskell)
Types are determined from the context of the reference. E.g.

```
fun double(x) = x*2;  (* ML knows x is int. *)
fun square(x) = x*x;  (* Error--ML can't tell. *)
Not error in OCaml because * and *. are separated.
```

Scoping

Scope of a [declaration of a] variable (or other identifier) = the parts of the whole program in which it can be used.

("Packages" and "modules" and separate compilation and the recent appearance of nested classes in C++ and Java add complexity to the notion of "lexical scope", but for now, we'll discuss only single-listing programs)

Lifetime or **Extent** of a variable = the time interval during execution in which the storage object for the variable is alive.

A storage object X is **born** when it is allocated, **dies** when it is no longer able to be used. . .and is "immortal" if it lives until the program exits.

Note that "dies" does not necessarily mean "is explicitly deallocated" (e.g., with dispose, "kill"), but means "can no longer be referenced." A dead storage object is called **garbage**.

Deallocation frees up the memory occupied by old or dead storage objects for use by other storage objects that may be dynamically created.

Deallocation may be explicitly invoked on a particular object that the programmer knows will not be used again (dispose or delete, etc.) Or:

Garbage Collection may be invoked to dispose of any dead objects that are taking up space, as done in Lisp, ML, and Java.

A storage object is **static** if it is born before execution (of the whole program) begins and dies when execution ends—i.e. if it is "immortal."

Global variables are always static. In Pascal, they are the only static variables. In COBOL, all variables are static. In C, variables within a block or subprogram (function) can be declared static:

```
int Foo (int a, b)
{                               Allocated before Foo is
    static int x = 0;           called, and initialized
                                to zero only once.
    x++                         This x is not visible
    printf (‘‘%d’’, x);         outside Foo.
}
```

So, in C, it's easy for a function to keep track of the number of times it is called, without risk of that value being altered elsewhere in the program. (Sebesta calls this function "sensitive to history.")

Dynamic storage objects are born during execution and may die during execution. Local variables, unless declared static in C, are dynamic storage objects—born on procedure call, die on termination of call. Dynamic objects can also be created by:

explicit allocation (e.g., Ada's new, C's malloc, calloc)
construction (e.g., cons in Lisp, :: in ML), in order to build up lists and other data types.

Two main storage allocation schemes:
Stack -- for objects with predictable nested lifetimes.
Heap -- for objects whose lifetimes are unpredictable, potentially immortal.

Stack Storage

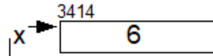
Define a "block" to be any segment of code in which local variables can be declared (e.g., {..} in C, block stmt. in Ada, procedures and functions in most PLs).

When a block is entered, an activation record (aka.
stack frame) is pushed onto the stack.

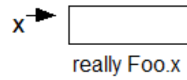
Def: The scope of a variable is the range of statements over which it is visible.
Def: The nonlocal variables of a program unit are those that are visible but not declared there. The scope rules of a language determine how references to names are associated with variables.

Static scope:
Based on program text
To connect a name reference to a variable, you (or the compiler) must find the declaration.
Search process: search declarations, first locally, then in increasingly larger enclosing scopes, until one is found for the given name. [This is done by following the static links until a definition of the variable is found. Often the compiler can pre-compute this.]
Enclosing static scopes (to a specific scope) are called its static ancestors; the nearest static ancestor is called a static parent.

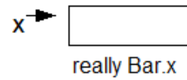
Variables can be hidden from a unit by having a "closer" variable with the same name. Ada, C++, and Java still allow access to these "hidden" variables using qualified ("dotted") names.
Dynamic Scope:
Based on the calling sequences of program units, not their textual layout (temporal versus spatial).
References to variables are connected to declarations by searching back through the chain of subprogram calls that forced execution to this point. [This is done by following the dynamic links until a definition for the variable is found. The compiler can usually never foretell this, so the traversing of links is all done at run-time.]




```
procedure Foo is
  x:float;
begin
  ...
end Foo;
```




```
procedure Bar is
  x:float digits 6 := 7.0;
begin
  ...
end Bar;
```




```
Main program
    X : INTEGER

    PROC Foo
      X : INTEGER
    End Foo

END Main
```

The inner occurrence of X is local to Foo.

Every time Foo is called, a storage object for “Foo.X” is born, and this storage object dies when Foo exits.